A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency while Preserving Responsiveness

Henry Cook* hcook@eecs.berkeley.edu
Miquel Moreto*† mmoreto@ac.upc.edu
Sarah Bird* slbird@eecs.berkeley.edu
Khanh Dao* khanhdao@berkeley.edu
David A. Patterson* pattrsn@eecs.berkeley.edu
Krste Asanovic* krste@eecs.berkeley.edu

*The Parallel Computing Laboratory
CS Division, EECS Department
University of California, Berkeley, USA

†Computer Architecture Department
Universitat Politecnica de Catalunya
Jordi Girona, 1-3, 08034, Barcelona, Spain

ABSTRACT

Computing workloads often contain a mix of interactive, latency-sensitive foreground applications and recurring background computations. To guarantee responsiveness, interactive and batch applications are often run on disjoint sets of resources, but this incurs additional energy, power, and capital costs. In this paper, we evaluate the potential of hardware cache partitioning mechanisms and policies to improve efficiency by allowing background applications to run simultaneously with interactive foreground applications, while avoiding degradation in interactive responsiveness. We evaluate these tradeoffs using commercial x86 multicore hardware that supports cache partitioning, and find that real hardware measurements with full applications provide different observations than past simulation-based evaluations. Co-scheduling applications without LLC partitioning leads to a 10% energy improvement and average throughput improvement of 54% compared to running tasks separately, but can result in foreground performance degradation of up to 34% with an average of 6%. With optimal static LLC partitioning, the average energy improvement increases to 12% and the average throughput improvement to 60%, while the worst case slowdown is reduced noticeably to 7% with an average slowdown of only 2%. We also evaluate a practical low-overhead dynamic algorithm to control partition sizes, and are able to realize the potential performance guarantees of the optimal static approach, while increasing background throughput by an additional 19%.

1. INTRODUCTION

Energy efficiency and predictable response times are first-order concerns across the entire computing spectrum, ranging from mobile clients to warehouse-scale cloud computers.

For mobile devices, energy efficiency is critical, as it affects both battery life and skin temperature, and predictable response times are essential for a fluid user interface. Some mobile systems have gone so far as to limit which applications can run in the background [1] to preserve responsiveness and battery life.

In warehouse-scale computing, energy inefficiencies increase electricity consumption and operational costs and can significantly impact the capital costs of the infrastructure needed to distribute power and cool the servers [2, 17]. As with mobile devices, researchers have also found unpredictable response times to be very expensive in warehouse-scale computing [31]. In one example, inserting delays in a search engine slowed down user response time by more than the delay, which decreased the number of overall searches, user satisfaction, and revenue; the results were so negative that the researchers stopped their experiment early [31]. To preserve responsiveness, cloud computing providers often dedicate large clusters to single applications, despite the hardware being routinely utilized at only 10% to 50% [2].

In this paper, we study the potential to reduce the waste from resources held idle in the name of responsiveness by co-scheduling background applications with foreground applications. Although some application workloads already include parallelized codes, few applications scale perfectly with increasing core count, leading to underutilized resources, and potentially providing an opportunity to increase system efficiency through consolidation. But this benefit can only be realized if there is minimal degradation in responsiveness of latency-sensitive user-facing tasks. In the client, the goal is to complete background work while the foreground task is active, so that the mobile device can quickly return to a very low-power hibernation mode, thereby extending battery life. In the cloud, the goal is to co-schedule background tasks, such as indexing, with user-facing web applications to obtain the greatest value from the huge sunk investment in machines, power distribution, and cooling.

Low-priority background tasks degrade the responsiveness of a high-priority foreground task primarily through contention for shared hardware resources, such as on-chip shared cache or off-chip DRAM bandwidth. In this pa-

ISCA '13 Tel Aviv, Israel

per, we study partitioning the capacity of a shared *last-level cache* (LLC) as a means to potentially mitigate the negative performance effects of co-scheduling, and design a lightweight practical algorithm to dynamically divide the LLC among applications. Our study uses a prototype commercial x86 multicore processor (Sandy Bridge) executing multiple large parallel applications taken from several modern benchmark suites, and we measure power and energy to explore the potential benefits of LLC partitioning. While other mechanisms to mitigate such degradation are the subject of active research [16, 23] (in particular, techniques for LLC partitioning [16, 20, 29, 33]), the past work has predominantly considered sequential applications and has mostly been simulation-based, which limits the size of possible workloads and the ability to accurately measure performance and energy. We found that our conclusions differ as a result of these differences in experimental methods.

In our analysis, we found significant potential for consolidation of applications without harming their responsiveness. The considerable size of the LLC (6 MB) makes cache partitioning unnecessary in many cases—a result not observed by previous work because the studies mostly simulated much smaller cache sizes (1–2 MB). Without any partitioning, we found that nearly 50% of the 45 benchmarks tested slowed down less than 2.5% with a co-scheduled background application. This is unsurprising considering that, when studied individually, we found that 44% of applications had working sets that fit in 1 MB and 78% fit in 3 MB. Overall, consolidation without LLC partitioning provided a 10% energy improvement and a 54% performance improvement. However, sharing did occasionally result in significant slowdowns, of up to nearly 35% in the worst case, with an average slowdown of 6%.

Co-scheduling applications with optimal static LLC partitioning increased the average energy improvement to 12% and the average performance improvement to 60%, while more effectively protecting the foreground application—the average slowdown was just 2% and the worst case only 7%. For 16% of the cases (almost exclusively those running with a low-scalability sensitive application), LLC partitioning provided even more significant speedups—20% on average.

We also introduce and evaluate a practical online dynamic partitioning algorithm, and show that it maintains foreground application performance to within 1% of the optimal static partition, while increasing the background computation throughput by 19% on average and as much as 2.5× in some cases.

2. EXPERIMENTAL METHODOLOGY

In this section, we describe the hardware platform and benchmarks used in our evaluation. While this paper does not evaluate server hardware, we believe the approach and resulting conclusions are just as relevant for the cloud, although the quantitative results would surely change.

2.1 Platform Configuration

We use a prototype version of Intel's Sandy Bridge x86 processor that is similar to the commercially available client chip, but with additional hardware support for way-based LLC partitioning. By using a real hardware prototype, we are able to run complete, full-sized applications for realistic time scales on a standard operating system and accurately measure performance and energy.

The Sandy Bridge client chip has 4 quad-issue out-of-order superscalar cores, each of which supports 2 hyperthreads using simultaneous multithreading [18]. Each core has private 32 KB instruction and data caches, as well as a private 256 KB non-inclusive L2 cache. The LLC is a 12-way set-associative 6 MB inclusive cache, shared by all cores via a ring interconnect. All cache levels are write-back.

The cache partitioning mechanism is way-based and works by modifying the cache-replacement algorithm. Each core can be assigned a subset of the 12 ways in the LLC. Way allocations can be completely private, completely shared, or overlapping. Although all cores can hit on data stored in any way, a core can only replace data in its assigned ways. Data is not flushed when the way allocation changes.

We use a customized BIOS that enables the cache partitioning mechanism, and run unmodified Linux-2.6.36 for all of our experiments. We use the Linux `taskset` command to pin applications to sets of hyperthreads.

2.2 Performance and Energy Measurement

To measure application performance, we use the `libpfm` library [11,27], built on top of the `perf_events` infrastructure introduced in Linux 2.6.31, to access performance counters available on the machine [19].

To measure on-chip energy, we use the energy counters available on Sandy Bridge to measure the consumption of the entire socket and also the total combined energy of cores, private caches, and the LLC. We access these counters using the Running Average Power Limit (RAPL) interfaces [19]. The counters measure power at a $1/2^{16}$ second granularity.

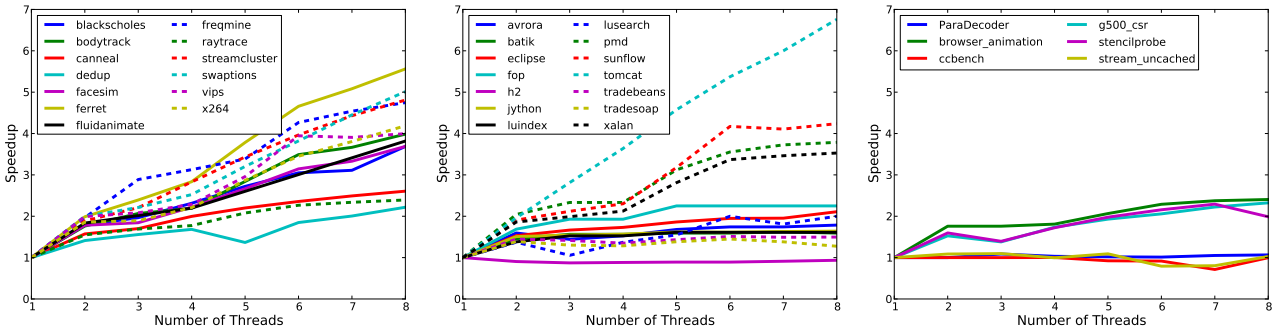
In addition, we use a FitPC external multimeter to measure the power consumed by the entire system at the wall socket with a 1 second granularity. We correlate the wall power data with the data collected from the hardware energy counters using time stamps. We observed less than one second of delay in these measurements consistently across all experiments. Together, these mechanisms allow us to collect accurate energy readings over the entire course of an application's execution.

2.3 Description of Workloads

We build our workload using a wide range of codes from three different popular benchmark suites: SPEC CPU 2006 [32], DaCapo [6], and PARSEC [4]. We include some additional applications to broaden the scope of the study, and microbenchmarks to exercise certain system features.

The **SPEC CPU2006** benchmark suite [32] is a CPU-intensive, single-threaded benchmark suite, designed to stress a system's processor, memory subsystem and compiler. Using the similarity analysis performed by Phansalkar et al. [28], we subset the suite, selecting 4 integer benchmarks (astar, libquantum, mcf, omnetpp) and 4 floating-point benchmarks (cactusADM, calculix, lbm, povray). Based on the characterization study by Jaleel [21], we also pick 4 extra floating-point benchmarks that stress the LLC: GemsFDTD, leslie3d, soplex and sphinx3. When multiple input sets and sizes are available, we pick the single *ref* input indicated by Phansalkar et al. [28]. SPEC CPU was the only benchmark suite used in many previous characterizations of LLC partitioning [16,29,33], but only a few hundred million instructions were simulated in these studies.

We include the **DaCapo** Java benchmark suite as a representative of a managed-language workloads. The managed



(a) PARSEC applications (b) DaCapo applications (c) Parallel applications and microbenchmarks

Figure 1: Normalized speed up as we increase the number of threads allocated to each application.

nature of the Java runtime environment has been shown to make a significant difference in some scheduling studies [12]. We use the latest 2009 release, which consists of a set of open-source, real-world applications with non-trivial memory loads, including both client and server-side applications.

The **PARSEC** benchmark suite is intended to be representative of parallel real-world applications [4]. PARSEC programs use various parallelization approaches, including data- and task-parallelization. We use the `pthreads` version for all benchmarks, with the exception of `freqmine`, which is only available in OpenMP. Although reduced input sets are available for simulation studies, we use the largest input sets designed for native execution. Previous characterizations of PARSEC have found it to be sensitive to cache capacity [4], but also resilient to performance degradation in the face of intra-application cache sharing [39], .

We add four **additional parallel applications** from local researchers to represent important algorithms at the core of future applications: `Browser_animation` is a multithreaded kernel representing a browser layout animation [25]; `G500_csr` code is a breadth-first search graph algorithm [3]; `Paradecoder` is a parallel speech-recognition application that takes audio waveforms of human speech and infers the most likely word sequence intended by the speaker [10]; `Stencilprobe` simulates heat transfer in a fluid using a parallel stencil kernel over a regular grid [22].

We also add two **microbenchmarks** that stress the memory system: `stream_uncached` is a memory and on-chip bandwidth hog that continuously brings data from memory without caching it, while `ccbench` explores arrays of different sizes to determine the structure of the cache hierarchy.

3. PERFORMANCE STUDIES

Our first set of experiments explores the sensitivity of each application to different resources in the system: hyperthreads, LLC capacity, prefetcher configurations, and the on-chip LLC bandwidth and off-chip DRAM bandwidth. We then use machine learning to cluster applications based on their resource requirements, and select a set of representative applications for further evaluation.

3.1 Thread Scalability

We first study parallel scalability for a fixed problem size. Figure 1 shows the speedup of each application as we increase its allocation from 1 to 8 threads. When adding new threads, we first assign both hyperthreads available in one core before moving on to the next core. For example, allocations with four threads correspond to running on both hyperthreads of two cores. This method fits our scenario of consolidating applications in a multiprogrammed environment, where different applications should be pinned to disjoint cores to avoid thrashing at inner cache levels [36].

Table 1: Summary of thread scalability

Suite	Low scalability	Saturated scalability	High scalability
PARSEC	–	canneal, dedup, raytrace	blackscholes, bodytrack, facesim, ferret, vips, x264, fluidanimate, freqmine, streamcluster, swaptions
DaCapo	h2, tradebeans, tradesoap	avrora, batik, eclipse, fop, jython, luindex, lusearch	pmd, sunflow, tomcat, xalan
SPEC	all	–	–
Parallel applications	paradecoder	browser_animation, g500, stencilprobe	–
μbenchmarks	ccbench, stream uncached	–	–

Many PARSEC applications scale well (Fig. 1a): six benchmarks scale up over 4×, four benchmarks between 3–4×, and just three show more modest scaling factors (2–3×). For the majority of these applications, we can see that performance keeps growing at a similar rate up to at least 8 threads. The DaCapo applications in Fig. 1b are largely less scalable than the PARSEC applications. Only two applications show speedups over 4×, with two others between 2–3×, and ten between 1–2.3×. Furthermore, the performance of all the low-scalability applications saturates after 4 or 6 threads. The intrinsic parallelism available in some of the DaCapo benchmarks together with the scalability bottlenecks for garbage collectors explain this behavior [15]. Finally, the scalability results for the additional parallel applications and microbenchmarks are presented in Figure 1c. The microbenchmarks are single-threaded, while the parallel applications are all memory-bandwidth-bound on this platform (we have observed parallel speedups on other platforms), explaining the limited scalability measured.

We now classify applications according to their thread scalability. Table 1 groups applications in each suite into three categories: applications with low scalability, applications that scale up to a reduced number of threads, and

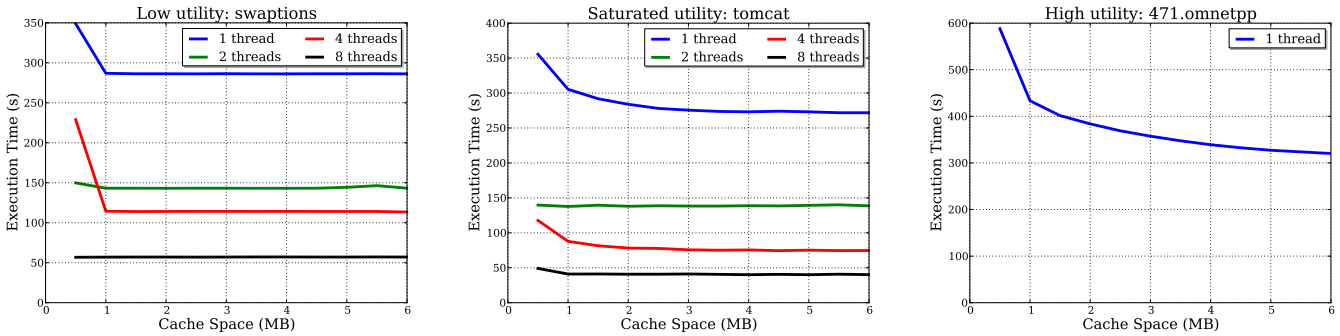


Figure 2: Applications representative of different LLC allocation sensitivities.

applications that continue to scale up with the number of threads. There are noticeable differences between suites, with PARSEC clearly being the most scalable.

3.2 Last-Level Cache Sensitivity

We next evaluate how sensitive the benchmarks are to amount of LLC capacity allocated to them. Taking advantage of the partitioning mechanism in the LLC, we change the LLC space allocated to a given application from 0.5 MB to 6 MB. In the interests of space, we show only the behavior of three representative applications in Figure 2.

Unsurprisingly, the first conclusion that we can draw is that running an application inside a 0.5 MB direct-mapped LLC is always detrimental. In addition to conflict misses from the direct mapping, inclusivity issues for inner cache levels lead to significant increases in execution time. The second observation is that the LLC is clearly overprovisioned for these applications. We found 44% of the applications only require 1 MB to reach their maximum performance, while 78% require less than 3 MB. Others have observed similar behavior for cloud computing applications [14].

Finally, we do not observe clear knees in execution time as we increase allocated LLC capacity. Previous simulation-based studies took advantage of these knees to propose dynamic cache partitioning techniques [26, 29, 33]. In contrast, performance improves smoothly with the allocated LLC capacity for all applications. The combination of memory-mapping functions, randomized LLC-indexing functions, pre-fetchers, pseudo-LRU eviction policies, as well as having multiple threads simultaneously accessing the LLC, serve to remove clear working-set knees in the real system.

Next, we classify applications into 3 categories according to their LLC sensitivity, ignoring the pathological direct-mapped 0.5 MB case. *Low utility* applications (*e.g.,* `swaptions`) yield the same performance despite increased available LLC space. *Saturated utility* applications (*e.g.,* `tomcat`) benefit from extra LLC space up to a saturation point. Finally, *high utility* applications (*e.g.,* `471.omnetpp`) always benefit from more LLC space. Figure 2 shows a representative application for each category. We observe that increasing the number of threads assigned to an application decreases LLC sensitivity. Additional cores result in larger aggregate private cache (L1 and L2) and a greater overlap of memory accesses, thereby reducing pressure on the LLC.

Table 2 categorizes the benchmarks in each suite according to their LLC utility. We highlight applications with more than 10 LLC accesses per kilo-instruction in bold, because these applications may cause bandwidth contention and pollute the LLC for other applications even if they do not benefit (in terms of execution time) from the allocated space. Overall, we find PARSEC applications have much more relaxed LLC requirements than the other suites. SPEC applications rarely require a large LLC despite having a significant number of LLC accesses.

Table 2: Summary of LLC allocation sensitivity.

Suite	Low	Saturated	High
PARSEC	blackscholes, bodytrack, dedup, ferret, fluidanimate, freqmine, raytrace, vips, **streamcluster**, swaptions,	**canneal**, facesim	**x264**
DaCapo	avrora, sunflow	batik, h2, jython, luindex, **tomcat**, **tradesoap**	eclipse, fop, lusearch, pmd, **tradebeans**, **xalan**
SPEC	436.cactusADM, **437.leslie3d**, **450.soplex**, 453.povray, 454.calculix, **459.GemsFDTD**, **462.libquantum**, **470.lbm**	**429.mcf**, 473.astar, 482.sphinx3	**471.omnetpp**
Parallel applications	–	**paradecoder**, stencilprobe	browser animation, **g500**
μbenchmarks	–	**ccbench**, **stream uncached**	–

3.3 Prefetcher Sensitivity

We now characterize the sensitivity of applications to hardware prefetching configurations, because some prefetchers are a shared resource that cannot be partitioned (unlike hyperthreads and LLC). In a multi-programmed environment, access streams from different applications could impact sensitive applications if they degrade prefetcher efficacy.

There are four distinct hardware prefetchers on Sandy Bridge platforms: 1) Per-core Data Cache Unit (DCU) IP-prefetchers look for sequential load history to determine whether to prefetch the data to the L1 caches; 2) DCU streamer prefetchers detect multiple reads to a single cache line in a certain period of time and choose to load the following cache lines to the L1 data caches; 3) Mid-Level cache (MLC) spatial prefetchers detect requests to two successive cache lines and are triggered if the adjacent cache lines are accessed; 4) MLC streaming-prefetchers work similarly to the DCU streamer-prefetchers, which predict future access patterns based on the current cache line reads. We can activate or deactivate each prefetcher by setting the corresponding machine state register (MSR) bits [19].

Figure 3 shows the execution time of the applications

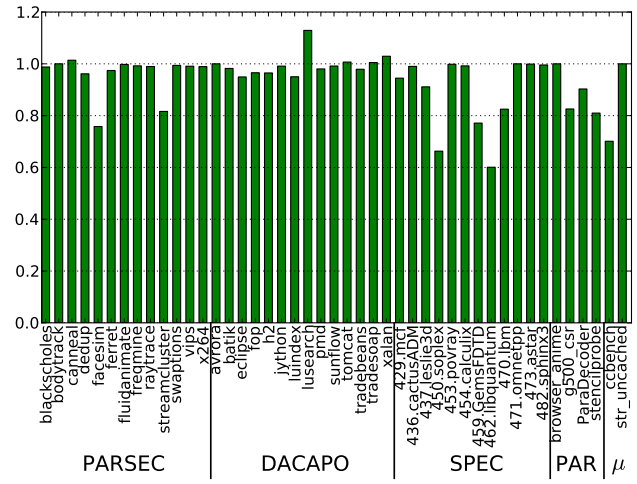


Figure 3: Normalized execution time when enabling all prefetchers enabled w.r.t. all prefetchers disabled.

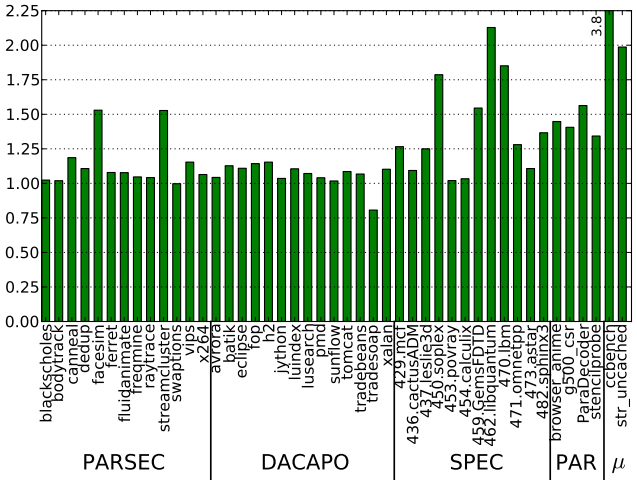


Figure 4: Increase in execution time when running with a bandwidth hog microbenchmark.

when all prefetchers are active normalized to the configuration with all prefetchers disabled. In general, the applications are more sensitive to the DCU spatial prefetcher, but the MLC prefetcher is also important for some applications. Nearly all applications are insensitive to the prefetcher configuration (36 out of 46). In PARSEC, only `facesim` and `streamcluster` benefit from the prefetchers. No DaCapo applications benefit significantly from the prefetchers, and `lusearch` performance even degrades when the prefetchers are active. In contrast, SPEC benchmarks are more sensitive to prefetching, particularly `450.soplex`, `459.GemsFDTD`, `462.libquantum` and `470.lbm`. The majority of the additional applications also benefit from the prefetchers. However, we also observed that increasing the number of threads in the system reduces overall prefetcher effectiveness.

3.4 LLC and DRAM Bandwidth Sensitivity

We also characterize how sensitive applications are to any memory bandwidth contention, because bandwidth is a shared resource that cannot be partitioned. Access

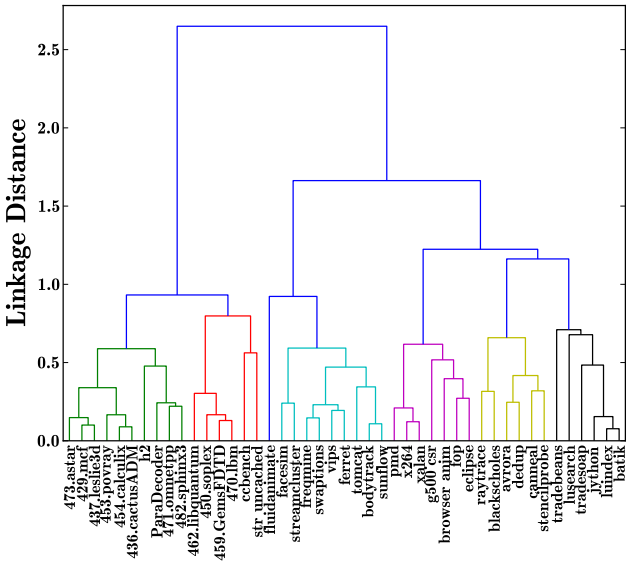


Figure 5: Clustering based on execution time, LLC space, memory bandwidth, and prefetcher sensitivity. All applications with the same color belong to the same cluster.

streams from concurrent applications could cause performance degradation of sensitive applications if they oversubscribe particular network links, memory channels, or Miss Status Handling Registers (MSHRs).

We characterize applications according to their performance when running together with a bandwidth-hogging microbenchmark (`stream_uncached`), which uses specially tagged load and store instructions to stream through memory without caching data in the LLC. Bandwidth-sensitive applications will suffer from being run concurrently with this benchmark. Figure 4 shows the increase in execution time of all applications when running with `stream_uncached`. Only two PARSEC applications suffer (`fluidanimante` and `streamcluster`), while DaCapo applications are not affected much by bandwidth contention. In the case of SPEC, some benchmarks are not affected at all (`436.cactusADM`, `453.povray`, `454.calculix`, `473.astar`) and others are heavily affected (`450.soplex`, `459.gemsFDTD`, `462.libquantum`, `470.lbm`). In contrast, all the added parallel applications are bandwidth sensitive. In general, the applications are more sensitive to bandwidth contention than to prefetcher configuration.

3.5 Clustering Analysis

To reduce our study to a feasible size, we use the application characterization studies described above to select a subset of the benchmarks representative of different application resource behaviors. Following in the footsteps of [28], we use machine learning to select representative benchmarks. We use a hierarchical clustering algorithm [28] with the *single-linkage* method from the Python library `scipy-cluster` .

We create a feature vector of 19 values for each application using the measurements from the previous subsections: 1) execution time as we increase the number of threads (7 features); 2) execution time as we increase the LLC size (10 features); 3) prefetcher sensitivity (1 feature); and 4) bandwidth sensitivity(1 feature). All metrics are normalized to the interval $[0, 1]$. The clustering algorithm finds the small-

Table 3: Cluster representatives

Suite	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6
PARSEC	–	–	**ferret**	x264	**dedup**	–
DaCapo	h2	–	sunflow	**fop**	avrora	**batik**
SPEC	**429.mcf**	**459.gems-FDTD**	–	–	–	–
Parallel Applications	Para-Decoder	–	–	browser animation	stencil-probe	–
μbenchmarks	–	ccbench	–	–	–	–

est Euclidean distance of a pair of feature vectors and forms a cluster containing that pair. It continues selecting the next smallest distance between a pair and forms another cluster. Linkage criteria can be used to adjust cluster formation. The single-linkage we selected uses the minimum distance between a pair of objects in different clusters to determine the distance between them.

Figure 5 shows the *dendogram* for the studied applications. The Y-axis represents the linkage-distance between applications. Applications within a distance of 0.9 are assigned the same cluster and colored to match. The first two clusters contain applications with low thread scalability. The first cluster is more sensitive to LLC space, but less sensitive to bandwidth and the prefetcher. Applications in the third cluster present high thread scalability and low cache utility and are insensitive to the prefetcher. The last three clusters are comprised of applications with saturated thread scalability, but different cache utility. The fourth cluster is more sensitive to cache space than the rest, the fifth is insensitive to cache space, and the sixth is insensitive to bandwidth contention. There is also a cluster with only one application (`fluidanimate`), which stands apart as it only runs correctly when allocated a power-of-2 number of threads. Due to this irregularity, we do not consider this cluster any further in our analysis. Table 3 lists representative applications by cluster for each benchmark suite. Applications highlighted in bold are closest to the centroid of the cluster. We select these applications to represent the cluster in our consolidation studies.

4. ENERGY VERSUS PERFORMANCE

Our next experiments explore the power, energy, and performance tradeoffs available in our system.

Controlling the number of cores assigned to an application, and the frequency at which those cores run, is the most well-studied technique to control energy consumption. However, it is worth noting that making energy-efficient optimizations sometimes involves counter-intuitive choices. For example, activating additional cores or raising frequency increases power consumption, but can often result in lower overall energy consumption per task, since the task may finish earlier allowing the system go into a much lower energy sleep state. This operating scenario is often described as *race-to-halt*, where the best energy efficiency is obtained by optimizing for the highest performance to more quickly complete a task and then move to a sleep state to save energy. However, allocating cores that do not improve performance can decrease energy efficiency. A memory-bound application is unlikely to see any performance benefit if run at a higher

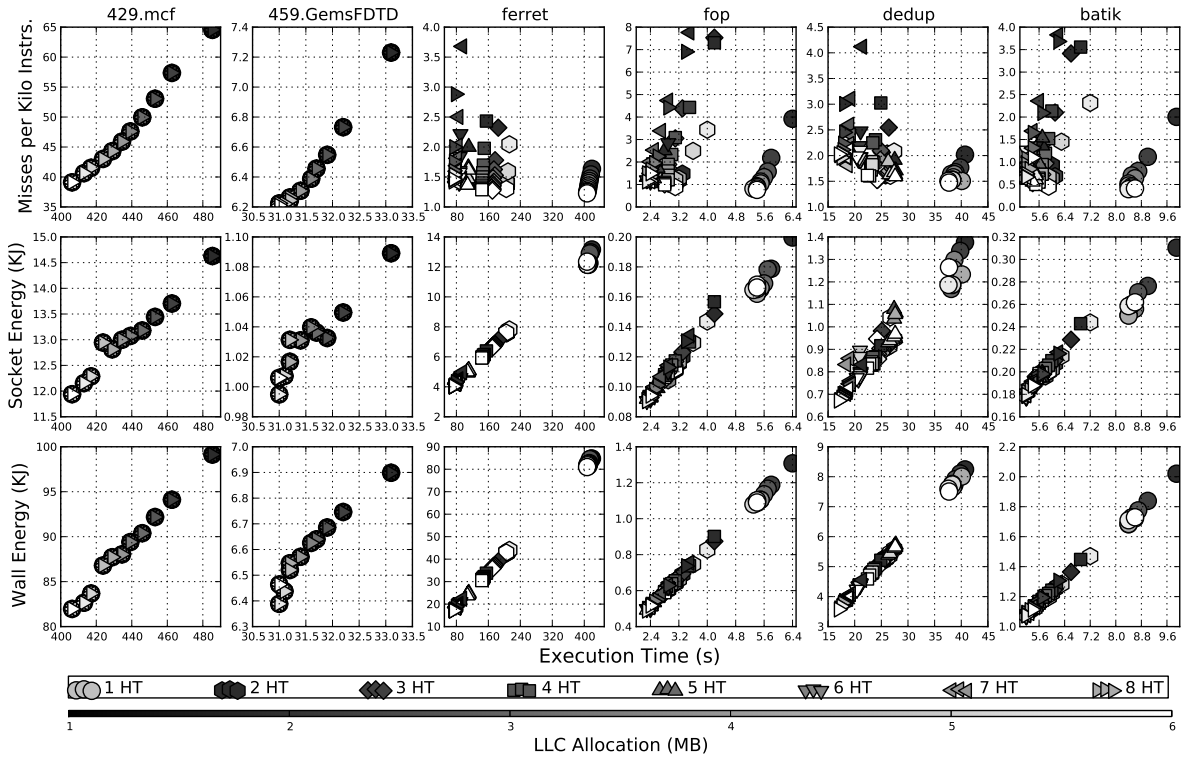


Figure 6: Example performance, energy and miss rate of resource allocations. Configurations with different numbers of hyperthreads are represented with different shapes. Shade represents cache allocation: darker configurations have a smaller cache allocation. All of the applications have multiple configurations that are optimal, indicating the potential for consolidation. The results also illustrate that *race-to-halt* is the optimal energy strategy.

frequency or allocated additional cores, but would consume more energy while waiting for data to be provided by the memory system.

Cache capacity allocation decisions are usually more straightforward, and typically only impact energy by changing the number of LLC misses an application incurs. LLC misses can increase energy consumption both by requiring additional data to be fetched from DRAM, and by the increased program runtime this might cause. Socket power does not change as a function of the cache allocated, since current hardware cannot turn off power to a portion of the cache. This limitation makes it desireable to try to reassign underutilized capacity to another application instead.

To better understand the space of possible performance and energy tradeoffs, we execute each representative with every possible thread and way allocation and measure the performance and energy. Each benchmark is tested with 1–8 threads and 1–12 cache ways (96 different allocations). Figure 6 shows plots of the runtime, LLC misses per kilo instruction (MPKI), and total socket and wall energy consumption of all possible resource allocations for the six cluster representatives. When considering execution time versus miss rate, we can see that some applications have runtimes that are tightly correlated with miss rate (`429.mcf` and `fop`), while others are insensitive (`ferret` and `dedup`), or see diminishing returns (`459.GemsFDTD` and `batik`).

When considering energy, our measurements strongly suggest that *race-to-halt* is the right optimization strategy for nearly all of our benchmarks. This is specially significant in the case of the wall energy, since the energy consumed in other parts of the system adds to total energy consumption. While there are a spread of points to consider when picking an allocation that minimizes LLC miss rate at a particular execution time, for nearly all benchmarks this curve narrows significantly when energy is factored in. We see this effect because in general miss rates are correlated with both increased energy and increased execution time, limiting the possibilities for a high-miss-rate allocation to have lower energy via a faster runtime, or a faster-runtime allocation to expend more energy than it saves.

We also can see many resource allocations achieve near optimal execution time, indicating that there should be spare resources available for background work to use when the application is running. Figure 7 illustrates this point by showing the contour plots of the wall energy for each benchmark. We obtained very similar figures when considering runtime and socket energy. Significantly, many applications have one or more energy-optimal configurations that are not the *largest* allocation, indicating that most applications do not require the whole system to achieve peak performance. Some applications do not benefit from more than one thread (`429.mcf` and `459.GemsFDTD`), others require all threads to minimize energy consumption (`dedup` and `ferret`), and some applications have a range of possible thread counts that maximize energy-efficiency (`batik` and `fop`). More importantly, all of them can yield some space in the LLC without affecting their performance, ranging from 0.5 MB (`429.mcf`) to 4 MB (`batik` and `ferret`). This resource gap between equally optimal allocations presents us with an opportunity: we could run additional work concurrently on the remaining LLC and core resources, assuming we can prevent destructive interference.

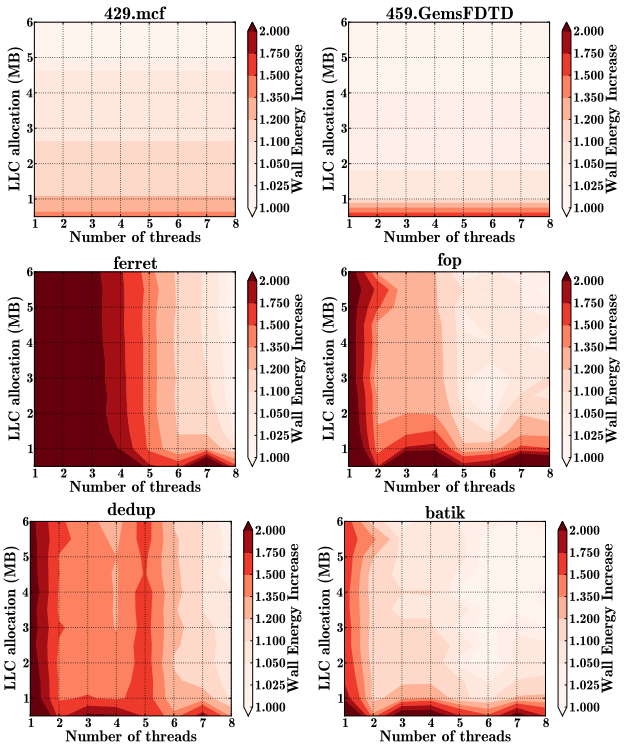


Figure 7: Wall energy contour plots for the cluster representatives. Darker colors represent higher energy consumptions.

5. MULTIPROGRAM ANALYSES

In this section, we show it is often possible to take advantage of the excess resources to save energy and improve system throughput, without impacting foreground application performance, by reducing the foreground application's allocation and running a concurrent background application. For some combinations of applications, we can effectively consolidate applications without partitioning the LLC. However, other combinations require LLC partitioning to protect the foreground application's performance. We examine the relative effectiveness of uneven, even and no cache partitioning in terms of energy, background throughput, and average and worst-case foreground performance degradation.

We run each multithreaded application with 4 threads on 2 cores with 2 active HTs. Some applications did exhibit slightly different cache scalability for 2, 4, 6, or 8 cores, so we used the 4-core values in our clustering analysis, ensuring a representative set of pairing scenarios for this study. We found that cases where applications were assigned odd numbers of hyperthreads (meaning that two applications were sharing a core) had many performance anomalies due to sharing of the L1 resources and thus chose to study splitting the cores evenly to more clearly observe the effects of consolidating applications in the LLC. Exploring alternative core allocations is the subject of future work [5].

5.1 Shared Cache Performance Degradation

To begin, we execute all possible pairs of applications together with no cache partitioning. In addition to LLC capacity, both applications are sharing the on-chip interconnection network to access the LLC and off-chip memory bandwidth. Figure 8 shows a heat map of the relative ex-

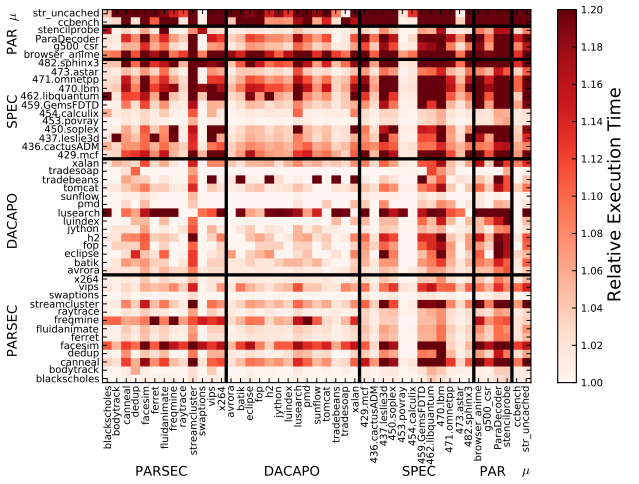


Figure 8: Normalized execution time of the foreground application (X-axis), when run concurrently with a background application (Y-axis). Spaces represented as a 20% slowdown may be greater than 20%

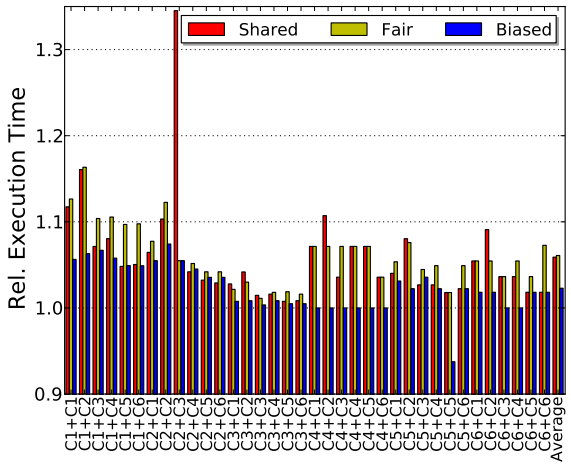


Figure 9: Effect of different consolidation approaches on the foreground application in the presence of different, continuously-running background applications. Normalized to the foreground application running alone.

ecution time of the foreground application for all possible pairs of applications. The values are normalized to the execution time of the application when running alone on the system with 2 cores with 2 active HTs each. For example, `canneal`'s execution time when running with `streamcluster` in the background increases 29% (dark red), while the execution time of `streamcluster` is affected much less (8.3%, bright red) when running with `canneal` in the background.

It is important to note that these relationships can be asymmetric. Some applications are *sensitive* to contention for the shared hardware resources (represented by a darker average vertical column). These benchmarks are more affected when running together with a background application. There is one sensitive application in PARSEC (`stream-cluster`), none in DaCapo, and 5 in SPEC CPU 2006 (`437.leslie`, `450.soplex`, `459.GemsFDTD`, `462.libquantum`, and `470.lbm`). All the new applications except `ccbench` are also sensitive. The average slowdown for these benchmarks is over 10%.

Independent of sensitivity, applications can also be *aggressive* and affect the performance of foreground applications if run in the background (presented by a darker average horizontal row). In our results, we find SPEC CPU 2006 and the additional parallel applications tend to be more aggressive. DaCapo benchmarks, on the other hand, only slightly affect other foreground applications. The applications that are the most significant aggressors (causing an average slowdown over 10%) are `canneal`, `lusearch`, `471.omnetpp`, `ParaDecoder`, `browser_animation` and `stream_uncached`.

Some applications are not affected when sharing the machine with a background application. Twenty-two of the 45 applications slow down less than 2.5% on average.

5.2 Mitigating Degradation with Partitioning

We consider three cache partitioning policies in this section. As the baseline, we use data from the previous subsection, where both applications share the entire cache without partitioning (*shared*). The other two approaches are to statically partition the cache between the two applications, in one case splitting the LLC evenly (*fair*), and in the second case giving each application some uneven allocation (*biased*). We evaluate all possible biased allocations and report results for the one that is the best (*i.e.,* among allocations with minimum foreground performance degradation, select the one that maximizes background performance).

Figure 9 presents the effectiveness of these three policies at preserving the performance of the representative applications. We run two applications simultaneously, each with 4 hyperthreads on two cores. All values are normalized to the execution time of the foreground application alone. For some foreground applications (C3, C5, C6), less than 5% degradation is present for the shared case, indicating their limited sensitivity. Thus, the improvement provided by partitioning is minor. For some applications pairs, degradation is somewhat or completely mitigated by biased partitioning, but not mitigated by fair partitioning. Overall, we find that biased partitioning results in lower average (2.3%) and worst case (7.4%) slowdown than shared (5.9% and 34.5% respectively). Half of the applications have effectively no slowdown with the biased partitioning as compared with only a quarter for shared. Fair is close to shared in terms of average slowdown (6.1%), but with a better worse case (16.3%).

The fact that degradation remains present in some cases, even though we have sequestered the applications on disjoint sets of cores and (in the biased case) provided them with optimally-sized cache allocations, implies that either the cache is just not large enough to accommodate both applications or bandwidth contention on the on-chip ring interconnect or off-chip DRAM interface is to blame. Prior work [23] has proposed mechanisms to partition such bandwidth resources, but unfortunately they are not available on extant hardware. While there is little we can do to mitigate the contention that exists on our platform, there are a number of points where energy optimization through consolidation is still possible.

We also examined more extreme cases with one foreground application and two or more copies of the background applications continuously running. However, adding additional applications only further increased contention for cache capacity and DRAM bandwidth. As expected the benchmarks already experiencing degradation with one background application, slowed down further when more were added.

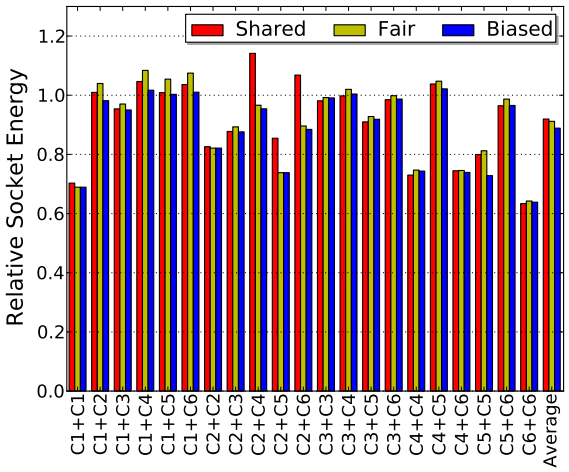


Figure 10: Socket energy values when running with a shared, evenly and optimally partitioned LLC normalized to the applications running sequentially on the whole machine. Consolidating applications results in an average energy improvement of 12% over running alone.

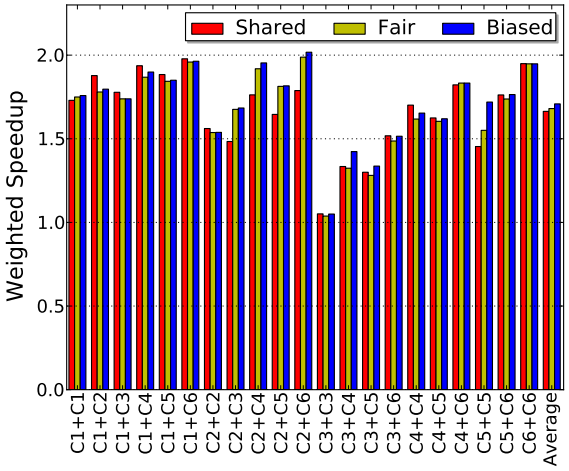


Figure 11: Relative performance of different partitioning strategies normalized to applications running sequentially on the whole machine. Consolidating applications results in an average speedup of 60% over running alone.

5.3 Energy Efficiency of Consolidation

Figure 10 compares running each application once, one after another, on the whole machine, with the different partitioning policies that run both applications once, concurrently, on the machine. The theoretical upper bound in energy savings is 50% and decreases as the applications have more disparate execution lengths. We measure an average energy improvement of 12% and a maximum of 37% for biased. The two other policies have similar but slightly lower improvements. The results differ mainly when `459.GemsFDTD` is running with `fop`, `dedup` or `batik`.

Figure 11 shows the weighted speedup of the application pairs running together as compared to each running alone. The results show an average speedup of 60% using biased partitioning and slightly lower for shared and fair. In 16% of the cases (mainly involving `459.GemsFDTD`), biased provides significant improvements over shared (20% on average).

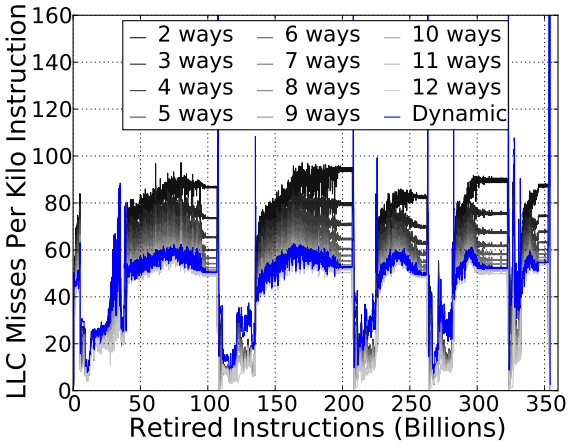


Figure 12: `429.mcf` LLC MPKI phase changes with different static and dynamic LLC allocations.

6. DYNAMIC CACHE PARTITIONING

In the previous section, we saw the potential for consolidation and found that for a static partition, biased partitioning provides the lowest average and worst-case foreground degradation, and the highest average performance and energy improvements. However, choosing the best partition would require testing all possible allocations, which is infeasible in practice, and such an optimal static allocation still cannot take into account phase-based behavior of the applications. In this section, we present our implementation of a dynamic cache-capacity allocation framework controlled by a utility-based policy to maximize background throughput and mitigate performance penalties without an oracle.

6.1 Opportunity

Applications often have phases with very different resource requirements. For example, Figure 12 shows the number of LLC misses per kilo-instruction (MPKI) for different cache allocations for `429.mcf`. This application transitions 5 times between low LLC MPKI and high LLC MPKI phases. In the phases with high MPKI, `429.mcf` requires 4.5 MB (9 ways) to reach 95% of its maximum performance, while in the other phases, only 1.5 MB (3 ways) are required. The phases with low MPKI represent opportunities to reduce the amount of LLC allocated to the application without negatively impacting its performance.

6.2 Phase-Detection Framework

We have created a software framework to monitor behavior and respond to phase changes by reallocating cache resources. We use `libpfm` to monitor the application's performance. The framework runs on the hyperthreads assigned to the application, and has negligible impact on application performance when passively monitoring behavior. The framework detects phase changes by looking for changes in LLC misses per kilo-instruction over a 100 millisecond interval. Algorithm 6.1 shows the phase-detection pseudocode.

6.3 Dynamic Partitioning Algorithm

When a phase change is detected, a dynamic reallocation algorithm is activated to determine the LLC capacity required by the new phase. Specifically, when the foreground

Algorithm 6.1: PHASE DETECTION ALGORITHM()

```
if not new_phase {
  if (|avg_MPKI-current_MPKI| >MPKI_THR1) {
    new_phase=1; /* static variable with initial value 0 */
    return 2; /* new phase just started */
  }
}
else if (|avg_MPKI-current_MPKI| <MPKI_THR2)
  new_phase = 0; /* phase change just finished */
return new_phase;
```

Algorithm 6.2: DYNAMIC CACHE PARTITIONING ALGORITHM()

```
if (phase_det()==2) {
  phase_starts=1;
  set_cache_to_6MB(fg)
}
else if (phase_det()==0 and phase_starts==1) {
  if (|last_MPKI-current_MPKI | < MPKI_THR3) {
    if (cache_allocated > 1MB)
      allocate_less_cache(fg);
    else phase_starts=0; /* Keep 1MB */
  }
  else {
    if (cache_allocated < 6MB)
      allocate_more_cache(fg); /* Keep previous allocation */
    phase_starts=0;
  }
}
last_MPKI = current_MPKI;
```

application starts or changes phase, the framework gives the application as much cache as possible (11 ways on our machine). The framework then gradually reduces the application's LLC allocation until negative performance effect is observed (MPKI goes up). The background application(s) are given the remaining LLC resources. On a reallocation, the data is not flushed since the partitioning mechanism only affects the replacement algorithm, which limits the performance overhead of reallocating. Algorithm 6.2 shows the pseudocode for our reallocation algorithm.

This algorithm uses several hysteresis effects in order to preserve the foreground application's performance. When dealing with applications with rapidly fluctuating MPKI rate, the framework will assign a LLC allocation that might be unnecessarily large. Additionally, data remaining in deallocated ways can hide the performance effects of the reallocation, allowing too much shrinkage. However, as soon as another application evicts the leftover data, a phase change will be detected and reallocation will return the foreground application to a suitable capacity.

A sensitivity study to set the MPKI derivative thresholds for phase detection and allocation size found selected parameters: $MPKI\_THR_1 = 0.02$, $MPKI\_THR_2 = 0.02$, and $MPKI\_THR_3 = 0.05$. We've found the results largely insensitive to small parameter changes.

The algorithm is applicable in cases where there are multiple background applications, as long as they can all be treated as peers. By pinning the background peers to certain cores, their accesses to the cache all become routed to the same partition, within which they contend for capacity. This extension does not affect the performance monitoring and cache reallocation performed on behalf of the foreground application. Supporting multiple latency-sensitive applications would require a more complex algorithm, as it is entirely possible for them to oversubscribe the cache, and in this case some component of the system would have to judge their relative utility [5]. We do believe that the performance monitoring aspect of our algorithm would be important to making such a judgement.

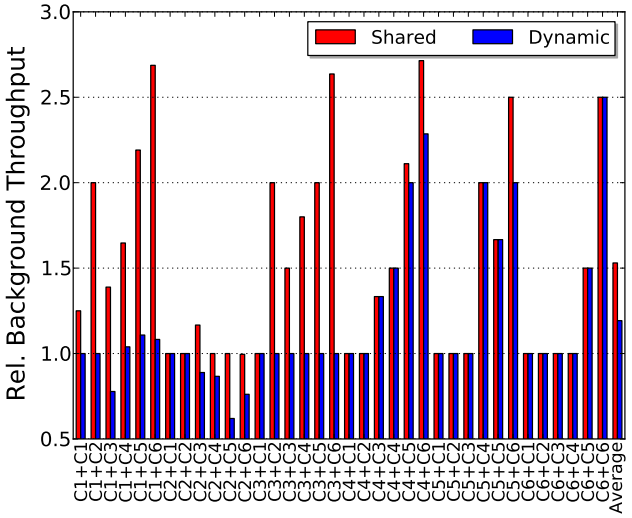


Figure 13: Summary of background rate improvement compared to the best static cache allocation for the foreground application. Our dynamic cache-partitioning approach improves background throughput an average of 19%. Sharing the cache improves throughput an average of 53%, but provides no performance isolation.

6.4 Efficacy

Across our benchmarks, we find that the framework is able to achieve foreground performance within 2% of the best static allocation. Figure 13 shows the improved background throughput as a result of the dynamic adaptation. In some cases we see significant throughput increases (up to 2.5x), resulting in a 19% throughput on average across all the pairs. For many cases, the limited number of phases in the foreground application or its high sensitivity to LLC allocation size does not allow for additional improvements over the best static policy. However, even in those cases the dynamic partitioning provides value, because it is a realizable way to achieve the performance of the optimal static allocation without application profiling. For comparison, using a shared LLC without partitioning results in a 53% improvement over the best static allocation. However, as shown before, this scenario can often result in significant performance loss (up to 35%) for the foreground application. As we saw in Section 4, performance improvements translate directly to energy improvements in our system, so we do not show the energy results.

While the dynamic partitioning mechanism does not always select the best cache capacity allocation for any one phase, overall it is still able to mitigate foreground degradation and increase background throughput. A suboptimal selection during some phases has minimal impact on foreground application latency, and overall the capacity allocated to the foreground application is smaller than when it is given its optimal static allocation for the entire runtime.

7. RELATED WORK

Several authors have evaluated way-partitioned caches with multiprogrammed workloads [16, 20, 29, 33] including

using partitioning to provide a minimum performance to applications [20,26]. However, these proposals were all evaluated on a simulator, and all but [20] used sequential applications — leading to different conclusions.

The 6 MB LLC in our system is often large enough for two applications to share without degradation, irrespective of cache partitioning policy. This behavior was not observed in previous studies based on simulators because the studies limited their cache sizes to 1–2 MB, which is closer to the working-set size of many of the studied applications.

Even in cases where degradation is present, LLC partitioning algorithms provide performance improvements for only a minority of our workloads. Consequently, relative to prior work, we measure reduced average performance improvements of LLC partitioning algorithms over naive sharing strategies (less than 3%). Simulation-based studies have reported much larger performance improvements [20,29,30], likely due to the reduced LLC capacity in their simulations, or possibly because they do not consider randomized LLC-indexing functions, memory-mapping functions, prefetchers, or memory-bandwidth contention.

Several groups have studied the impact of workload consolidation in shared caches [36,37,39] for datacenter applications. They do not explore hardware solutions such as cache partitioning, or client-side workloads.

Other authors make use of *page coloring* to partition the LLC by sets [9, 24, 34]. Cho and Lin [9] experiment with page coloring on a simulator to reduce access latency in distributed caches. Tam et al. [34] map pages into each core's private color using static decisions with Linux on a POWER5. Lin et al. [24] evaluate a dynamic scheme using page coloring on an Intel Xeon 5160 processor. However, there is a significant performance overhead inherent to changing the color of a page. Another challenge is that the number of available partitions is a function of page size, so increasing page size can make page coloring ineffective or impossible [9, 24, 34]. The experiments on real machines use only 2 running threads and make decisions only every several seconds. In contrast, our approach can change LLC partitions much more quickly and with minimal overhead.

Others have proposed hardware support to obtain the cache miss rate, IPC curves or hardware utilization information [7, 26, 29] to provide QoS. These approaches require hardware modifications and will not work on current processors. Chanda et al. [8] explored on-line methods to extrapolate these curves, but their approach has significant complexity and overhead. Tam et al. [35] use performance counters on POWER5 to predict miss curves for different page coloring assignments. They use a stack simulator and an LLC address trace to obtain the miss curve and statically assign colors. Others have measured LLC misses to identify thrashing applications [38] or predict contention [13] enabling the scheduler to pick co-running applications with more affinity. Xie and Loh [38] further use the LLC measurements to partition the cache according to their classification of applications as thrashing or non-thrashing. LLC misses can also be used to change the replacement policy of the LLC to partition the cache at finer granularity [30].

8. DISCUSSION AND CONCLUSIONS

This paper measures the energy and performance of an LLC-partitioning scheme on real x86 hardware using a diverse set of full-sized serial and parallel applications. These measurements show that optimizing for performance is still also the optimal strategy for energy optimization, but that a significant number of applications do not need to run on all the available cores or with all the LLC capacity to reach an energy-optimal execution point. Based on this characterization, we evaluated the potential to co-schedule different applications using LLC partitioning and CPU binding techniques. We succeed in significantly reducing energy consumption (12% on average) and increasing performance (60% on average), while minimizing the performance degradation of the foreground applications (2% on average and 7% worst case) using LLC partitioning.

We found that for around half of our workloads, cache partitioning is unnecessary: most parallel applications do not need more than 1 MB LLC to reach their maximum performance. Consequently, the 6 MB LLC in our system (and other current systems) is typically enough for two applications to share without degradation irrespective of cache partitioning policy. Simulation studies using cache sizes closer to working set sizes (1–2 MB), show excessive interference and hence greater potential improvement from cache partitioning (>10%).

We have also frequently been told by industry partners that cache partitioning will be less effective than shared caching because shared caching naturally adjusts the allocation for each application dynamically, whereas cache partitioning does not let other applications reclaim unused resources. Overall, while we find that naive LLC sharing can often be effective, there can be a significant downside in foreground performance degradation (35% worst case). Although responsiveness may not matter in some domains, it has become an increasingly crucial factor in the user-experience for both mobile and warehouse-scale computing [1,2,17,31]. Additionally, we found that cache partitioning was actually more effective than shared caching on average: across the entire suite of applications, it provided increased performance (60%) and energy (12%) improvements versus shared caching (54% performance improvement and 10% energy) without the potential performance degradation. The worst case was only 7%, and average foreground degradation was just 2%. Furthermore, LLC partitioning algorithms present significant performance improvements (20% on average) in 16% of workloads (workloads comprised of sensitive and aggressive LLC applications).

Static partitioning is impractical, as it would require profiling of all applications before deployment and cannot react to application phases or input-dependent behavior. We found that a simple dynamic adjustment algorithm provided an effective consolidation solution. Our lightweight online dynamic partitioning algorithm maintained foreground performance to within 2% of the best static partitioning, while further increasing the background computation throughput 19% on average and by as much as 2.5× in some cases.

Finally, we determined that partitioning or other quality-of-service mechanisms for memory bandwidth could potentially be a further effective hardware addition to consider on future systems. All of the worst-case foreground slowdowns with cache partitioning (and without) were from the applications shown to be the most sensitive to memory bandwidth. The slowdowns occurred even when the background application did not have high bandwidth demands — implying that in order to to achieve robust performance isolation, latency quality-of-service in particular would need to improve.

9. ACKNOWLEDGEMENTS

We would especially like to thank everyone at Intel who made it possible for us to use the cache-partitioning machine in this paper, including Opher Kahn, Andrew Herdrich, Ravi Iyer, Gans Srinivasa, Mark Rowland, Ian Steiner and Henry Gabb. We would also like to Scott Beamer, Chris Celio, Shoaib Kamil, Leo Meyerovich, and David Sheffield for allowing us to study their applications. Additionally, we would like to thank our colleagues in the Par Lab for their continual advice, support, and, feedback. Research supported by Microsoft (Award 024263) and Intel (Award 024894) funding and by matching funding by U.C. Discovery (Award DIG07-10227). Additional support comes from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung. M. Moreto was supported by the Spanish Ministry of Science under contract TIN2012-34557, a MEC/Fulbright Fellowship, and by an AGAUR award (BE-DGR 2010).

10. REFERENCES

[1] Apple Inc. iOS App Programming Guide. `http://developer.apple.com/library/ios/DOCUMENTATION/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf`.

[2] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[3] S. Beamer, K. Asanovic, and D. A. Patterson. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. Technical Report UCB/EECS-2011-117, EECS Department, University of California, Berkeley, Nov 2011.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] S. Bird, B. Smith, K. Asanović, and D. A. Patterson. PACORA: Dynamically Optimizing Resource Allocations for Interactive Applications. Technical report, University of California, Berkeley, April 2013.

[6] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[7] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable Performance in SMT Processors: Synergy between the OS and SMTs. *IEEE Trans. Computers*, 55(7):785–799, 2006.

[8] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, pages 340 – 351, 2005.

[9] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *MICRO*, pages 455–468, 2006.

[10] J. Chong, G. Friedland, A. Janin, N. Morgan, and C. Oei. Opportunities and challenges of parallelizing speech recognition. In *HotPar*, 2010.

[11] S. Eranian. Perfmon2: a flexible performance monitoring interface for linux. In *Ottawa Linux Symposium*, pages 269–288, 2006.

[12] H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley. Looking back and looking forward: power, performance, and upheaval. *Commun. ACM*, 55(7):105–114, July 2012.

[13] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2):49–57, 2010.

[14] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ASPLOS*, pages 37–48, 2012.

[15] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. In *PLOS*, pages 1–5, 2011.

[16] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO*, 2007.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[18] Intel Corp. Intel 64 and ia-32 architectures optimization reference manual, June 2011.

[19] Intel Corp. Intel 64 and ia-32 architectures software developer's manual, March 2012.

[20] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, pages 25–36, 2007.

[21] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation – a pin-based memory characterization of the spec cpu2000 and spec cpu2006 benchmark suites. Technical report, VSSAD, Intel Corporation, 2007.

[22] S. Kamil. Stencil probe, 2012. `http://www.cs.berkeley.edu/~skamil/projects/stencilprobe/`.

[23] J. W. Lee, M. C. Ng, and K. Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *ISCA*, pages 89–100, 2008.

[24] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, pages 367 –378, feb. 2008.

[25] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik. Parallel schedule synthesis for attribute grammars. In *PPoPP*, 2013.

[26] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. FlexDCP: a QoS framework for CMP architectures. *SIGOPS Oper. Syst. Rev.*, 43(2):86–96, 2009.

[27] Perfmon2 webpage. perfmon2.sourceforge.net/.

[28] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *ISCA*, pages 412–423, 2007.

[29] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO*, pages 423–432, 2006.

[30] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *ISCA)*, June 2011.

[31] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity*, 2009.

[32] Standard Performance Evaluation Corporation. SPEC CPU 2006 benchmark suite. `http://www.spec.org`.

[33] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *HPCA*, pages 117–128, 2002.

[34] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared l2 caches on multicore systems in software. In *WIOSCA*, 2007.

[35] D. K. Tam, R. Azimi, L. Soares, and M. Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *ASPLOS*, pages 121–132, 2009.

[36] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, pages 283–294, 2011.

[37] C.-J. Wu and M. Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *ISPASS*, pages 2–11, 2011.

[38] Y. Xie and G. H. Loh. Scalable shared-cache management by containing thrashing workloads. In *HiPEAC*, pages 262–276, 2010.

[39] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *PPoPP*, pages 203–212, 2010.